INRIA

CENTRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél. 954 90 20

Rapports de Recherche

N° 142

METAL, UN LANGAGE DE SPÉCIFICATION POUR LE SYSTÈME MENTOR

Bertrand MÉLÈSE

Juin 1982

METAL, un langage de spécification pour le système MENTOR

METAL, a specification language for the MENTOR system

Bertrand Mélèse

I.N.R.I.A
Domaine de Voluceau
B.P. 105
78153 Le Chesnay

Mots clés: MENTOR, environnement de programmation, éditeur syntaxique, syntaxe abstraite, construction applicative d'arbres .

Key words: MENTOR, programming environment, syntax directed editor, abstract syntax, applicative tree building.

Résumé:

Le système MENTOR est maintenant un système générique. Nous entendons par là que MENTOR n'est plus figé autour de deux langages déterminés (Pascal et Mentol), mais qu'il est possible, pour un utilisateur non spécialiste, de définir lui-même le formalisme (langage) qu'il désire manipuler.

Le langage Métal, présenté ici, est le langage dans lequel un utilisateur doit définir son formalisme pour l'intégrer dans le système MENTOR. Un programme Métal, PM, est donc la définition d'un certain formalisme F. La compilation de ce programme crée MENTOR-F, c'est à dire une instance de MENTOR apte à manipuler les programmes F.

Dans la première partie de ce papier, nous décrivons les principaux éléments de ce nouvel aspect du système MENTOR, et nous montrons, à travers un exemple, comment un utilisateur doit s'y prendre pour définir un formalisme en Métal. Dans la deuxième partie nous donnons une idée de l'organisation de la «machine abstraite de construction d'arbres» qui est le processeur de MENTOR chargé de la construction des arbres durant l'analyse syntaxique des programmes.

Mots clés: MENTOR, environnement de programmation, éditeur syntaxique, syntaxe abstraite, structure d'arbres. décompilateur,

Abstract:

The MENTOR system is now a generic system. This means that MENTOR is no longer frozen around two languages (Pascal and Mentol), and includes facilities allowing a non-specialist user to define his own formalism (i.e. language) to be manipulated with MENTOR.

We present here the language Metal which is used to define a formalism to be integrated into MENTOR. A program written in Metal is the definition of such a formalism F; the compilation of this program produces MENTOR-F, i.e. an instance of MENTOR that manipulates programs written in F.

In the first part, we describe the main aspects of this new development of MENTOR and illustrate, by means of an example, how a user can define his own formalism in Metal. In the second part, we describe shortly the «abstract tree processor», which is the processor of MENTOR that builds trees during parsing of programs.

Key words: MENTOR, programming environment, syntax directed editor, abstract syntax, unparser, tree structure.

METAL

Un langage de spécification pour le système

MENTOR

Bertrand Mélèse

I.N.R.I.A
Domaine de Voluceau, 78153 Le Chesnay

Table des matières:

Introduction

Le système MENTOR [DON 80, DON 79, DON 75, MEL 80, MEL 81, MUL 80] est un environnement de programmation interactif, multi-formalismes et extensible. Le coeur du système est un éditeur guidé par la syntaxe du formalisme qu'il manipule à un instant donné. Il crée et utilise une représentation des données sous forme d'arbres étiquetés: les arbres de syntaxe abstraite. En MENTOR-Pascal, par exemple, les données manipulées sont des fragments

de programmes Pascal représentés par des arbres conformes a la syntaxe abstraite de ce langage [MUL 80].

Le systeme MENTOR est piloté par un langage de programmation spécialisé dans la manipulation d'arbres: le langage MENTOL. Ce langage est indépendant du formalisme manipulé. En MENTOL, l'utilisateur de MENTOR peut programmer des transformations et des calculs en tous genres sur les arbres qu'il manipule. Un certain nombre d'outils ont déjà été écrits autour de MENTOR-Pascal [MEL 80, MEL 81, MOR 79, SCH 82].

Le système MENTOR est développé en Pascal sous MENTOR-Pascal. Actuellement, il tourne sur cinq (grosses) machines différentes: MULTICS, IRIS 80, VAX, PDP 10, IBM. Les compilateurs Pascal qui existent sur chacune de ces machines sont sensiblement différents les uns des autres. Le transport du système a toujours été fait en écrivant des procédures MENTOL de transformation de programmes Pascal, pour effectuer les modifications nécessaires au passage de ces programmes d'un compilateur a un autre, prouvant ainsi que le transport des programmes est une application importante d'un systeme tel que MENTOR. Nous envisageons, dans un avenir proche, l'implémentation de MENTOR sur un micro-ordinateur a base de MOTOROLA 68000 et sous UNIX [BSTJ 78].

Le système MENTOR est un systeme générique de manipulation de formalismes (langages), ce qui signifie qu'il ne dépend d'aucun langage particulier. Les données relatives à un langage particulier sont contenues dans des tables (fichiers) extérieures au systeme. Ces tables sont chargées au moment où l'utilisateur indique le langage sur lequel il désire travailler. Grâce à cette organisation, la taille du systeme est indépendante du nombre de formalismes manipulables à un moment donné. De plus, l'adjonction, la suppression ou la modification d'un des formalismes n'a aucune répercussion sur les autres et ne demande aucune intervention dans le système lui-même.

Le langage **METAL** est le langage de spécification du systeme MENTOR. L'intégration d'un nouveau formalisme, ou langage de programmation, au systeme est faite par l'écriture d'un programme METAL définissant ce formalisme. Le langage METAL existe lui-même sous MENTOR. L'utilisateur qui desire intégrer un nouveau langage au systeme dispose donc de l'environnement MENTOR-METAL pour mettre au point le programme METAL qui définit le langage de ses rêves. En MENTOR-METAL la commande de compilation **COMPILE** crée les tables du formalisme décrit par le programme METAL manipulé. MENTOR peut ensuite être appelé immédiatement sur ce nouveau formalisme. Rappelons que le passage d'un formalisme a un autre peut être fait dynamiquement au cours d'une session MENTOR: il suffit pour cela de demander le chargement des tables du nouveau formalisme. Ces tables sont alors empilées sur les anciennes auxquelles on pourra revenir ensuite par une commande de dépilage. On retrouvera alors la session originale dans l'état où elle était restée. La combinaison des deux derniers points laisse entrevoir la puissance de l'environnement disponible pour la définition et la mise au point de nouveaux formalismes: aussitôt un nouveau formalisme F créé, MENTOR-F peut être invoqué «par dessus» la session MENTOR-METAL dans laquelle la mise au point du programme METAL définissant F a lieu. Si le MENTOR-F créé n'est pas satisfaisant, on revient dans la session METAL, on modifie le programme et on le compile à nouveau créant ainsi la nouvelle version de MENTOR-F que l'on peut invoquer aussitôt.

L'implantation du langage METAL sous MENTOR a été faite par «bootstrap», c'est a dire par la compilation du programme METAL qui définit le langage METAL. Le compilateur

METAL travaille EXCLUSIVEMENT à partir de l'arbre de syntaxe abstraite qui représente le programme sous MENTOR-METAL. L'utilisateur de MENTOR qui désire définir un nouveau langage est donc vivement encouragé à le faire directement à l'aide de MENTOR-METAL qui, de plus, lui fournira un environnement de programmation autour du langage METAL. Notons ici que le langage METAL est un exemple de langage implanté sous MENTOR et possédant un compilateur intégré à MENTOR.

Un programme METAL, PM, qui défini un formalisme F, contient les éléments suivants:

1- La **syntaxe concrète** de F, c'est à dire un ensemble de règles permettant de décider si une phrase donnée est admise par le formalisme F. Ces règles sont écrites dans une forme proche de la BNF. L'analyseur syntaxique de F sera dérivé à partir de ces règles.

2- La **syntaxe abstraite** de F, c'est a dire la syntaxe des arbres qui représenteront les formules légales (les programmes légaux) dans le formalisme (langage) F. La correspondance entre la forme textuelle et la forme arborescente d'un même objet est assurée par deux processeurs: **l' analyseur-constructeur** qui construit l'arbre a partir de la forme textuelle, et le **décompilateur** qui produit une représentation textuelle à partir de l'arbre de syntaxe abstraite.

3- Les **fonctions de construction** des arbres de syntaxe abstraite. Ces fonctions sont le lien logique entre la forme textuelle, ou concrète, d'une phrase de F et sa forme arborescente, ou abstraite. Elles indiquent, pour chaque construction du langage, quel est l'arbre associé. A partir de ces fonctions sera dérivé le constructeur d'arbres pour le formalisme F. L'analyseur syntaxique de F sera ensuite relié au constructeur d'arbres par une interface indépendante du langage, pour former l'analyseur-constructeur associé a F.

   En d'autres termes, à chaque production de la syntaxe concrète de F est associée une action sémantique, ou fonction de construction d'arbre. Chaque fois qu'une réduction est effectuée, au cours de l'analyse syntaxique d'un programme F, l'action sémantique correspondante est exécutée, un arbre est donc construit et est passé a la suite du processus d'analyse.

4- Les **fonctions de décompilation** de F. Ces fonctions sont le lien logique entre la forme arborescente et la forme textuelle. Elles indiquent le morceau de texte associé a chaque arbre élémentaire. A partir de ces fonctions sera construit le **décompilateur** de F, c'est a dire le processeur qui produit une représentation textuelle à partir d'un arbre abstrait.

Parmi les quatre points précédents, le seul qui demande un réel travail de conception à un nouvel utilisateur est la définition de la syntaxe abstraite de son langage. En effet, les points 3 et 4 découlent de la syntaxe abstraite de façon d'autant plus simple que celle-ci est mieux conçue. De plus, une syntaxe abstraite cohérente et naturelle facilitera énormément l'écriture ultérieure de procédures Mentol pour manipuler les arbres qui représentent des programmes F.

Indiquons que METAL est capable de traiter des gros langages: E. Morcos a introduit le langage Ada sous MENTOR en écrivant un programme METAL qui définit ce langage. Actuellement, les langages disponibles sous MENTOR sont: Pascal, Ada, METAL, Flip et Mentol. Pour Pascal il existe un environnement de programmation très complet sous MENTOR [MEL 80, MEL 81]. Pour les autres, les environnements sont plus jeunes mais se développent peu à peu. Signalons également l'existence d'un Pascal «transformable», a partir duquel un utilisateur peut, très rapidement, définir des extensions de ce langage. Nous pensons que la prochaine grande étape dans le développement de MENTOR sera sa transformation en un système «multi-langages», c'est à dire en un système dans lequel il sera possible de manipuler *simultanément* plusieurs programmes dans plusieurs langages différents, ouvrant ainsi des espoirs de traductions automatiques d'un langage dans un autre.

Dans la première partie de cet article nous décrivons les principales structures du langage METAL en utilisant comme exemple un petit langage: le langage Flip [KAHN 81]. Un manuel complet du langage METAL fait l'objet d'une autre publication [MEL 82]. Dans la deuxième partie, nous expliquons le fonctionnement de la **machine abstraite de construction d'arbres** , qui est le processeur chargé de la de création des arbres durant l'analyse syntaxique d'un programme ou d'un fragment de programme.

I-LE LANGAGE METAL

Un programme METAL est structuré en chapitres et en sections. Le nom du langage défini est donné dans l'en-tête du programme. La syntaxe abstraite du langage défini est donnée dans les sections «abstract syntax». Dans les sections «rules» on trouve les productions de la syntaxe concrète du langage défini sous une forme très proche de la BNF. A chaque production est associée une fonction de construction d'arbre qui indique comment est construit l'arbre associé à la partie gauche de la production en fonction des arbres associés aux non-terminaux de la partie droite de cette production. Dans la plupart des cas les terminaux (mots clés, symboles de ponctuations, parentheses dans les expressions) qui apparaissent dans une partie droite de production n'ont pas à être pris en compte dans les fonctions de construction d'arbres puisqu'ils n'apparaitront pas dans l'arbre de syntaxe abstraite. En effet, l'information contenue dans les noeuds de l'arbre est en generale suffisante pour que le décompilateur soit capable de re-créer lui même ces terminaux. Le lecteur de reportera a l'Appendice, dans lequel nous donnons le programme METAL complet qui definit le langage Flip, pour se faire une idée de la forme d'un programme METAL.

Dans cette première partie, nous suivons le cheminement logique qu'un utilisateur devrait respecter lorsqu'il définit un nouveau langage en METAL. Nous commençons donc par définir la syntaxe concrete de Flip, nous définissons ensuite sa syntaxe abstraite, puis nous montrons comment écrire les fonctions de construction d'arbres associées a chacune des productions.

I.1-La syntaxe concrète de Flip

Le langage **Flip** est un formalisme de description de transparents pour rétro-projecteurs qui a été défini par G. Kahn. L'exécution d'un programme Flip produit le transparent décrit par ce programme, sur un traceur de courbes.

La première chose à faire, pour intégrer un nouveau langage à Mentor, est de définir la syntaxe concrète de ce langage, c'est à dire l'automate capable de décider si une phrase donnée appartient au langage ou pas. La syntaxe concrète de Flip, exprimée dans une forme proche de la BNF, est la suivante:

```
<FLIP>          ::= <COULEUR>
                  |<CHAINE>
                  |<LISTE_LIGNES>
                  |en <COULEUR> ecrire <FLIP>
                  |en <COULEUR> encadrer <FLIP> fincadre
                  |pivoter <FLIP>
                  |horiz <LISTE_BANDES> fhoriz
                  |vertic <LISTE_BANDES> fvertic
                  |couvrir <LISTE_FLIP> fcouvrir ;
<LISTE_FLIP>    ::= <FLIP>
                  |<LISTE_FLIP> avec <FLIP> ;
<LISTE_BANDES>  ::= <BANDE>
                  |<LISTE_BANDES> <BANDE> ;
<BANDE>         ::= <NOMBRE> : <FLIP> ;
<LISTE_LIGNES>  ::= <LIGNE>
                  |<LISTE_LIGNES> <LIGNE> ;
<COULEUR>       ::= %COULEUR ;
<CHAINE>        ::= %CHAINE ;
<NOMBRE>        ::= %NOMBRE ;
<LIGNE>         ::= %LIGNE ;
```

Les identificateurs précédés par le symbole «%», tels que %COULEUR désignent des unités lexicales génériques définies par des expressions régulières dans l'analyseur lexical du langage Flip. Les couleurs (%COULEUR) sont des identificateurs en majuscules, les chaînes (%CHAINE) sont des suites de caractères entre apostrophes, les lignes (%LIGNE) sont des suites de caractères commençant par le symbole «-» et finissant par le symbole de fin de ligne usuel. Les nombres sont des entiers sans signes.

I.2- Exemple de programme Flip

```
horiz
     10 :
          en ROUGE encadrer
               en NOIR ecrire
                    -En MENTOR-Flip
                    -on fait
                    -des beaux flips
          fincadre
     20 :
          en BLEU ecrire
               -Les beaux transparents
               -font
               -les beaux exposes
fhoriz
```

I.3-Définition de la syntaxe abstraite de Flip

La syntaxe abstaite de Flip est la syntaxe qui décrit la représentation arborescente des programmes Flip, qui sera manipulée par MENTOR. La syntaxe abstraite d'un langage est composée d' **opérateurs** et de **phylums:**

> Les **opérateurs** seront les noeuds de l'arbre. Ils sont d'arité fixe (inférieure à 3) ou d'arité variable (noeuds de liste). Les opérateurs d'arité nulle sont les feuilles de l'arbre et correspondent aux atomes du langage. Les opérateurs d'arité fixe (non nulle) peuvent avoir des fils de types différents, tandis que les fils d'un noeud de liste doivent tous être de même type. Le type d'un noeud est le **phylum** auquel ce noeud appartient.
>
> Les **phylums** *(Phylum: du grec phulon «race, tribu», (petit Robert))* sont simplement des ensembles, non vides, d'opérateurs. A chaque emplacement (ou occurence) d'un arbre de syntaxe abstraite est associé un phylum qui indique quels sont les opérateurs qui ont le droit de se trouver à cet emplacement. Le phylum associé à un emplacement donné ne dépend que du père de cet emplacement. Les phylums permettent à MENTOR de maintenir des arbres corrects, c'est à dire des arbres qui, une fois décompilés, correspondront à des programmes syntaxiquement corrects. Ainsi, chaque fois qu'une opération modifiant un arbre est exécutée, MENTOR vérifie la légitimité de cette opération et ne l'exécute que si elle est effectivement correcte, c'est à dire si l'arbre qui en résulte est un arbre correct. La notion de phylum permet donc d'introduire des restrictions sur l'ensemble des arbres qu'il est possible de construire à partir d'un ensemble d'opérateurs donné.

Définir une syntaxe abstraite signifie donc tout simplement:

1- décider quels sont les opérateurs avec, pour chacun d'eux, leur arité et les phylums de leurs fils;
2- définir les phylums.

Ces deux opérations sont bien sûr étroitement mélées et sont faites simultanément. Il est à noter que, pour la plupart des langages existants, seule la syntaxe concrète existe et est plus ou moins standardisée. Pour ces langages, une syntaxe abstraite doit être définie et il n'y a pas de façon mécanique de le faire. Nous pensons cependant que les critères suivants devraient être observés, dans la mesure du possible:

Les sous arbres devraient correspondre à des concepts sémantiquement significatifs du langage traité.

Les structures abstraites devraient être suffisamment proches de la syntaxe concrète pour que l'utilisateur les comprenne et s'en souvienne facilement. Ainsi, l'ordre des objets lors d'un parcours en préordre de l'arbre abstrait devrait être le même que dans la syntaxe concrète du langage.

Les manipulations courantes dans le langage considéré devraient être faites facilement sur l'arbre. Ceci sera en général une conséquence du premier point.

Bien entendu, la syntaxe abstraite devrait être la plus simple possible. Ceci est souvent en contradiction avec le premier et le deuxième point car les syntaxes concrètes comportent souvent des particularités qui ont peu à voir avec les concepts sémantiques.

La syntaxe abstraite de Flip est donnée ci-dessous dans le format accepté par le langage Métal: les opérateurs sont en minuscules, les phylums sont en majuscules. Pour définir un opérateur on donne les phylums de ses fils, et pour définir un phylum on indique les opérateurs qui lui appartiennent. L'arité d'un opérateur est égale au nombre de phylums dans la partie droite de la règle le définissant. Les symboles «+ ...» indiquent que l'opérateur correspondant est un opérateur de liste. Par exemple, si on regarde la syntaxe abstraite de Flip ci-dessous, on observe que:

l'opérateur «cadre» est binaire, son premier fils est un opérateur appartenant au phylum «COULEUR» tandis que son deuxième fils est un opérateur appartenant au phylum «FLIP».

l'opérateur «vertic» est un opérateur de liste dont tous les fils appartiennent au phylum «BANDE». Le symbole «+» utilisé indique que cette liste ne peut pas être vide. On utilise «*» au lieu de «+» pour définir des listes qui peuvent, éventuellement, être vides.

l'opérateur «chaine» est nullaire: ce sera une feuille de l'arbre.

```
couleur         -> ;
chaine          -> ;
nombre          -> ;
ligne           -> ;
pivot           -> FLIP;
fonte           -> COULEUR FLIP;
cadre           -> COULEUR FLIP;
bande           -> NOMBRE FLIP;
horiz           -> BANDE + ... ;
```

```
vertic          -> BANDE + ... ;
flips           -> FLIP + ... ;
lignes          -> LIGNE + ... ;

FLIP            ::=couleur chaine lignes fonte
                   cadre pivot horiz vertic flips;
BANDE           ::=bande;
COULEUR         ::=couleur;
CHAINE          ::=chaine;
NOMBRE          ::=nombre;
LIGNE           ::=ligne;
```

I.4-Le programme Métal définissant MENTOR-Flip

Une fois en possession d'une syntaxe concrète et d'une syntaxe abstraite, nous sommes prêts à écrire le programme Métal définissant MENTOR-Flip. Le coeur d'un programme Métal est une liste de productions à chacune desquelles on associe une fonction de construction d'arbre. Nous allons écrire le programme Métal définissant MENTOR-Flip, et que nous appellerons Flip.Métal.

a. Construction des arbres atomiques

Les arbres atomiques, qui seront les feuilles de l'arbre final, sont créés par le constructeur «atom». Ce constructeur prend en opérande gauche le nom de l'opérateur atomique à construire et en argument la valeur de cet atome. La valeur peut être immédiate, c'est alors une chaîne de caractères entre doubles apostrophes, ou bien obtenue à partir de la valeur d'une unité lexicale. Considérons la production suivante:

```
<COULEUR>       ::= %COULEUR ;
```

Une réduction effectuée par cette production, au cours de l'analyse syntaxique d'un programme Flip, signifie qu'une unité lexicale, répondant à la définition d'une couleur, a été rencontrée dans le texte analysé. L'arbre correspondant doit être un noeud nullaire avec l'opérateur «couleur» de la syntaxe abstraite de Flip et, comme valeur, la valeur de l'unité lexicale. La fonction qui construit cet arbre est la suivante:

```
couleur-atom(%COULEUR)
```

Dans le programme Métal nous aurons donc la règle

```
<COULEUR>       ::= %COULEUR ;
    couleur-atom(%COULEUR)
```

qui indique en plus que l'arbre, une fois construit, sera associé au non-terminal <COULEUR>, c'est à dire au non-terminal qui constitue la partie gauche de la production, pour être transmis à la suite du processus. Pour la construction des atomes des arbres Flip nous aurons donc les quatre règles suivantes dans le programme Flip.Métal:

```
<COULEUR>          ::= %COULEUR ;
    couleur-atom(%COULEUR)
<CHAINE>           ::= %CHAINE ;
    chaine-atom(%CHAINE)
<NOMBRE>           ::= %NOMBRE ;
    nombre-atom(%NOMBRE)
<LIGNE>            ::= %LIGNE ;
    ligne-atom(%LIGNE)
```

b. Construction des opérateurs d'arité fixe (non nulle)

Considérons maintenant la production

```
<FLIP>             ::= pivoter <FLIP> ;
```

La fonction de construction d'arbre associée est

```
pivot(<FLIP>)
```

ce qui indique que l'on construit un arbre unaire dont l'opérateur est «pivot» et dont le fils est l'arbre associé au non-terminal <FLIP> dans le contexte où l'on exécute cette fonction. Comme dans les règles précédentes, l'arbre construit est ensuite associé au non-terminal qui forme la partie gauche de la production pour être transmis à la suite du processus.

Tous les noeuds d'arité fixe sont construits sur le même principe que les noeuds unaires. Le nombre d'arguments doit simplement correspondre à l'arité du noeud. Voyons tout de même un petit exemple de construction d'arbre binaire. Prenons la production

```
<FLIP> ::= en <COULEUR> encadrer <FLIP> fincadre ;
```

dont la fonction associée sera:

```
cadre(<COULEUR>,<FLIP>)
```

ce qui signifie que l'arbre construit est l'opérateur «cadre», qui est binaire, avec comme premier fils l'arbre associé au non-terminal <COULEUR>, et, comme deuxième fils, l'arbre associé au non-terminal <FLIP>. L'arbre construit est ensuite associé au non-terminal <FLIP> pour être passé à la suite du processus.

Les règles qui construisent les opérateurs d'arité fixe dans le programme Flip.Métal seront donc les suivantes:

```
<FLIP>             ::= en <COULEUR> ecrire <FLIP> ;
    fonte(<COULEUR>,<FLIP>)
<FLIP>             ::= en <COULEUR> encadrer <FLIP>
                         fincadre ;
    cadre(<COULEUR>,<FLIP>)
<FLIP>             ::= pivoter <FLIP> ;
    pivot(<FLIP>)
```

```
<BANDE>          ::= <NOMBRE> : <FLIP> ;
    bande(<NOMBRE>,<FLIP>)
```

c. Construction des opérateurs de liste

Voyons maintenant comment nous allons construire les noeuds de liste. Pour cela, considérons les deux productions de la grammaire de Flip qui définissent une liste de lignes:

```
<LISTE_LIGNES>   ::= <LIGNE> ;
<LISTE_LIGNES>   ::= <LISTE_LIGNES> <LIGNE> ;
```

Lorsque l'analyseur passe sur la première, il faut construire une liste de lignes contenant un élément, tandis que lors d'un passage sur la deuxième il faut ajouter un élément à une liste existante. La fonction de construction d'arbre associée à la première production est

```
lignes-list((<LIGNE>))
```

Le constructeur «list» indique que l'opérateur construit est un opérateur de liste. Son opérande gauche est le nom de cet opérateur, et son argument est la liste à construire. Ici, la liste à construire n'a qu'un seul élément qui est l'arbre associé au non-terminal <LIGNE>. La fonction associée à la deuxième production est

```
lignes-post(<LISTE_LIGNES>,<LIGNE>)
```

Le constructeur «post» prend en opérande gauche un opérateur de liste, en premier argument une liste dont l'opérateur doit être le même, et en deuxième argument un élément qui sera ajouté à la fin de la liste. Ici, la liste est l'arbre associé au non-terminal <LISTE_LIGNES>, et l'élément est l'arbre associé au non-terminal <LIGNE>. Finalement, pour construire les listes de lignes nous avons les deux règles suivantes:

```
<LISTE_LIGNES>   ::= <LIGNE> ;
    lignes-list((<LIGNE>))
<LISTE_LIGNES>   ::= <LISTE_LIGNES> <LIGNE> ;
    lignes-post(<LISTE_LIGNES>,<LIGNE>)
```

Il existe un constructeur «pre» qui s'utilise de façon similaire à «post», mais en inversant l'ordre des arguments, et qui a pour effet de rajouter un élément en tête de la liste. Dans ce papier nous en resteront volontairement aux utilisations les plus simples. Le lecteur pourra trouver des compléments d'informations dans le manuel Métal.

Intéressons nous maintenant à la construction des listes de bandes. Vous avez probablement remarqué qu'il y a deux opérateurs qui sont des listes de bandes dans la syntaxe abstraite de Flip: les opérateurs «horiz» et «vertic». Donc, au moment de construire une liste de bandes, on ne sait pas encore si l'opérateur sera «horiz» ou «vertic». On ne le saura que lorsque l'analyseur syntaxique passera sur l'une des deux productions

```
<FLIP>           ::= horiz <LISTE_BANDES> fhoriz ;
<FLIP>           ::= vertic <LISTE_BANDES> fvertic ;
```

Nous allons donc construire systématiquement les listes de bandes avec l'opérateur «horiz», par un choix arbitraire, et nous renommerons la liste par la suite dans les cas où il fallait l'opérateur «vertic». Les règles pour construire les listes de bandes sont donc

```
<LISTE_BANDES>  ::= <BANDE> ;
    horiz-list((<BANDE>))
<LISTE_BANDES>  ::= <LISTE_BANDES> <BANDE> ;
    horiz-post(<LISTE_BANDES>,<BANDE>)
```

tandis que la construction d'un noeud «horiz» devient tout simplement

```
<FLIP>          ::= horiz <LISTE_BANDES> fhoriz ;
    <LISTE_BANDES>
```

Dans cette dernière règle, la fonction de construction d'arbre est réduite à un non-terminal de la partie droite de la production. C'est la façon, en Métal, de transmettre un arbre au reste du processus sans rien faire d'autre. En effet, l'arbre associé au non-terminal <LISTE_BANDES> se retrouve associé, sans modification, au non-terminal <FLIP>. Pour construire l'opérateur «vertic» nous écrirons en revanche la régle suivante:

```
<FLIP>          ::= vertic <LISTE_BANDES> fvertic ;
    vertic-list(<LISTE_BANDES>)
```

Ici, l'argument du constructeur «list» n'est plus une liste, mais un non-terminal qui désigne une liste. Dans ce cas, la liste est reconstruite avec l'opérateur qui se trouve en opérande gauche. L'effet d'une telle règle est donc simplement le renommage de la liste associée au non-terminal <LISTE_BANDES>, qui devient une liste dont l'opérateur est «vertic». Du point de vue de l'utilisateur, le renommage d'une liste peut être vu comme le changement de l'opérateur sans changer ses fils.

A ce point, nous avons tout ce qu'il faut pour écrire le programme Flip.Métal. Ce programme est donné en appendice.

d. Indexation des non-terminaux

Il est possible de faire référence aux non-terminaux de la partie droite d'une production dans n'importe quel ordre et, si nécessaire, plusieurs fois, dans les fonctions de construction d'arbres. Si, dans la partie droite d'une production le même non-terminal apparait plusieurs fois, on distingue ses occurences en les indiçant par des entiers comme le montre l'exemple suivant:

```
<ADDITION>          ::= <EXP> + <EXP> ;
    plus(<EXP>.1,<EXP>.2)
```

I.5-Les instructions Métal «let» et «case»

Les techniques de construction d'arbres que nous avons vues jusqu'ici ne sont pas suffisantes, en général, pour résoudre tous les cas qui se présentent lors de la définition d'un formalisme en Métal. Les instructions «let» et «case» vont nous permettre de faire des tests et de décomposer les arbres déjà construits pour réutiliser certains de leurs composants dans la construction de nouveaux arbres. Le principe de ces deux instructions est de fournir le filtrage (en anglais: «matching») comme opération de base du langage Métal.

a. L'instruction «case»

Reprenons par exemple les deux productions suivantes de la syntaxe de Flip

```
<FLIP>          ::= en <COULEUR> ecrire <FLIP>
                  |en <COULEUR> encadrer <FLIP> fincadre
```

et réécrivons les d'une autre manière, équivalente syntaxiquement:

```
<TEXTE_OU_CADRE> ::= ecrire <FLIP>
                   |encadrer <FLIP> fincadre
<FLIP>           ::= en <COULEUR> <TEXTE_OU_CADRE>
```

Une manière d'écrire les fonctions de construction d'arbres pour obtenir les mêmes arborescences qu'auparavant serait alors la suivante:

```
<TEXTE_OU_CADRE> ::= ecrire <FLIP> ;
     fonte(couleur,<FLIP>)
<TEXTE_OU_CADRE> ::= encadrer <FLIP> fincadre ;
     cadre(couleur,<FLIP>)
<FLIP>           ::= en <COULEUR> <TEXTE_OU_CADRE> ;
     case <TEXTE_OU_CADRE>
          when fonte(X,Y)
               => fonte(<COULEUR>,Y)
          when cadre(X,Y)
               => cadre(<COULEUR>,Y)
     end case
```

Dans les deux premières règles, nous construisons les opérateurs «fonte» et «cadre», sans connaitre leur premier fils. Ici, nous leur avons mis l'opérateur «couleur» en premier fils pour rappeler que c'est celui-ci qui est attendu à cet endroit. En fait, le premier fils construit ici ne sera jamais utilisé et n'apparaitra pas dans l'arbre final. On peut donc y mettre ce que l'on veut. La seule chose indispensable est que l'arité des opérateurs construits, «fonte» et «cadre», soit respectée. Dans ces deux règles, nous avons donc construit des arbres binaires dont, moralement, le premier fils est vide. Dans les deux cas, cet arbre est associé au non-terminal <TEXTE_OU_CADRE>.

Dans la troisième règle, l'arbre associé au non-terminal <TEXTE_OU_CADRE> va servir de sélecteur à l'instruction «case». Il sera donc filtré successivement par les schémas apparaissant dans les alternatives du «case» jusqu'à ce qu'un filtrage réussisse. La fonction de construction

d'arbre qui se trouve derrière la flèche dans l'alternative correspondante sera alors exécutée et l'arbre renvoyé par cette fonction sera l'arbre renvoyé par l'instruction «case».

Ici le «case» a deux alternatives. Une alternative est de la forme

```
when  schéma
   => fonction
```

Nous appelons **schéma** une expression qui représente un arbre constant, c'est à dire indépendant d'une exécution particulière du programme Métal. Un schéma peut contenir des **méta-variables** (identificateurs en majuscules) qui seront instanciées au cours du filtrage. Dans la fonction de construction d'arbre qui apparait en partie droite d'une alternative, la référence à une méta-variable permet de réutiliser l'arbre qui lui a été associée par le filtrage précédent.

Dans l'instruction «case» précédente, l'arbre associé au non-terminal <TEXTE_OU_CADRE> est d'abord filtré par le schéma «fonte(X,Y)». En cas de succès, les méta-variables «X» et «Y» sont associées respectivement au premier et au deuxième fils de cet arbre. Ensuite la fonction de construction d'arbre «fonte(<COULEUR>,Y)» est exécutée. En cas d'échec de ce premier filtrage, l'arbre associé au non-terminal <TEXTE_OU_CADRE> est filtré par le schéma de la deuxième alternative: «cadre(X,Y)». Si un nouvel échec intervient, l'instruction «case» échoue. En cas de succès, les méta-variables «X» et «Y» sont instanciées comme dans le cas précédent et la fonction «cadre(<COULEUR>,Y)» est exécutée.

b. L'instruction «let»

Reprenons à nouveau les deux mêmes productions de la syntaxe de Flip, et réécrivons les encore une fois d'une autre manière, équivalente syntaxiquement:

```
<FLIP>          ::= <EN_COULEUR> ecrire <FLIP>
                  |<EN_COULEUR> encadrer <FLIP> fincadre
<EN_COULEUR>    ::= en <COULEUR>
```

Pour obtenir les mêmes arbres que prècèdemment, il faudrait alors écrire les fonctions de construction d'arbres comme suit:

```
<EN_COULEUR> ::= en <COULEUR> ;
    fonte(<COULEUR>,null_tree)
<FLIP>       ::= <EN_COULEUR> ecrire <FLIP> ;
    let fonte(X,Y) = <EN_COULEUR>
        in fonte(X,<FLIP>)
<FLIP>       ::= <EN_COULEUR> encadrer <FLIP>
                 fincadre ;
    let fonte(X,Y) = <EN_COULEUR>
        in cadre(X,<FLIP>)
```

Dans la première règle, nous construisons l'opérateur «fonte» dans tous les cas car, à ce moment là, il est encore impossible de savoir si l'opérateur final sera «fonte» ou «cadre». Seul le premier fils est significatif mais ici, par coquetterie, nous avons utilisé un nouvel opérateur, «null_tree», comme deuxième fils. Celui-ci devrait donc, simultanément être rajouté,

en tant qu'opérateur nullaire, dans la syntaxe abstraite de Flip. Dans les deux règles suivantes, les «let» commencent par filtrer l'arbre associé au non-terminal <EN_COULEUR> par le schéma «fonte(X,Y)», ce qui a comme effet d'associer à la méta-variable «X» le premier fils de cet arbre et à la méta-variable «Y» son deuxième fils. Ensuite, la fonction de construction d'arbre qui apparait après le mot clé «in» est exécutée et l'arbre renvoyé par cette fonction est l'arbre renvoyé par le «let». Durant l'exécution de cette fonction, la référence à une méta-variable permet de réutiliser l'arbre qui lui a été associé par le filtrage précédent.

A ce niveau, vous vous dites probablement que les instructions «let» et «case» servent à redresser la situation en cas de productions écrites sans réfléchir. C'est un peu le cas en effet mais, parfois, on ne peut pas faire autrement, en particulier si la syntaxe concrète d'un langage a été définie sans penser pas du tout à la construction des arbres. De plus, on est parfois amené à modifier les productions de syntaxe concrète pour les adapter aux caractéristiques de l'analyseur général utilisé. La plupart des analyseurs généraux exigent par exemple que la grammaire définie soit une grammaire LALR1. En tous cas, il ressort des exemples précédents que, pour faciliter l'écriture des fonctions de construction d'arbres, il faut s'efforcer d'écrire les productions de telle sorte qu'à chacune d'entre elles puisse correspondre la construction d'un sous arbre complet.

I.6-Décompilateur

Les fonctions de décompilation ne sont pas encore implémentées dans le langage Métal et nous n'en parlerons donc pas dans cet article. Disons simplement que le principe de ces fonctions sera comparable à celui des fonctions de construction d'arbres.

Dans l'état actuel du système MENTOR, le décompilateur d'un nouveau langage doit être écrit en Pascal. Il existe un squelette standard de décompilateur qui permet la construction rapide d'un décompilateur décent. Notons par ailleurs qu'il existe dans MENTOR un décompilateur universel qui est capable de décompiler n'importe quelle structure d'arbre de manière canonique, ni très jolie ni très compacte mais utilisable, par exemple, pour la mise au point d'un nouveau langage.

I.7-Analyseurs syntaxiques et analyseurs lexicaux

Pour fonctionner, MENTOR a besoin d'un analyseur général. Sur le site Multics de l'I.N.R.I.A., l'analyseur-constructeur général utilisé est le système SYNTAX développé à l'I.N.R.I.A. par le groupe Langages et Traducteurs. Cependant, les conditions que MENTOR demande à un analyseur général sont assez faibles et il doit être possible de l'utiliser avec d'autres analyseurs généraux:

Il faut que MENTOR puisse, d'une part, récupérer les unités lexicales au fur et à mesure qu'elles se présentent dans le texte en entrée (y compris les commentaires). D'autre part, MENTOR a besoin de savoir quelle production a été utilisée chaque fois qu'une réduction a été faite par l'analyseur syntaxique, pour pouvoir exécuter la fonction de construction d'arbre appropriée.

II-LA MACHINE ABSTRAITE DE CONSTRUCTION D'ARBRES

MENTOR, au cours de l'analyse syntaxique d'un programme ou d'un fragment de programme, construit l'arbre abstrait associé par l'intermédiaire d'une M.A.d.C.A. Lorsque l'analyseur syntaxique lui passe le contrôle, cette M.A.d.C.A. exécute un programme qui a été créé par la compilation des fonctions de construction d'arbres qui, dans le programme METAL sont associées à chacune des productions. Ce processeur fonctionne exactement comme une machine dont les instructions élémentaires sont des instructions de construction d'arbres et dont la mémoire est structurée de manière à manipuler des arborescences de façon naturelle. Cette machine simulée fonctionne à l'aide d'une pile dans laquelle sont stockés les fragments d'arbres déjà construits.

Nous décrivons succinctement ci-dessous quelques unes des instructions élémentaires de la machine de construction d'arbres. Un document complet à ce sujet est en cours de préparation.

II.1-Les instructions de construction d'arbres

Le premier groupe d'instructions de construction d'arbres est constitué par des instructions qui prennent en argument le nom de l'opérateur à construire, vont chercher sur la pile les sous arbres à utiliser, et empilent le résultat de leur construction. Un exemple typique est l'instruction décrite ci-dessous:

MK2 op : Construction d'un arbre binaire avec l'opérateur «op». Le sommet de pile est pris comme premier fils et est dépilé. Le nouveau sommet de pile est pris comme deuxième fils et est dépilé à son tour. L'arbre résultat est empilé.

Sur le modèle de MK2, il existe des instructions pour la constructions des arbres nullaires unaires et ternaires (MK0, MK1, MK3), pour la construction des listes (MKLST, RNLIST) et pour la construction des atomes (MKATOM). Pour la construction des listes il y a en plus les instructions POST et PRE qui rajoutent un élément à la fin (POST) ou au début (PRE) d'une liste existante, la liste et l'élément étant tous deux pris sur la pile. Pour la construction des atomes génériques, nous utilisons l'instruction GPS:

GPS op : Construction d'un arbre atomique avec l'opérateur «op» et, comme valeur, la dernière unité lexicale passée par l'analyseur lexical à l'analyseur syntaxique.

II.2-Les instructions de filtrage d'arbres

MATCH n : Effectue le filtrage de l'arbre contenu dans le registre de filtrage par le schéma numero n. Cette instruction renvoie «vrai» ou «faux» dans le registre logique selon le résultat du filtrage. Si le schéma contient des méta-variables un environnement est créé, dans lequel à chaque méta-variable est associé le sous arbre par lequel elle a été instanciée au cours du filtrage. Dans la mémoire de la machine, une zone spéciale est

réservée au stockage des schémas qui sont addressés pas un simple numéro d'ordre.

LOAD : Charge le registre de filtrage avec le sommet de pile et dépile.

GIMETA n : Empile l'arbre qui a été associé à la méta-variable numéro n dans l'environnement créé par le filtrage précédent.

II.3-Les instructions de manipulation de la pile

PD : Dépile

PU n : Empile le contenu du registre numéro n. Actuellement la machine fonctionne avec 9 registres aptes à contenir des arbres temporaires.

INI n : Charge les n premiers registres (N<=9) avec les n positions supérieures de la pile en les dépilant.

II.4-Autres instructions

La machine possède en outre une instruction d'arrêt (STOP) et des instructions de saut: JP, JPT, JPF. Pour les 2 dernières, la valeur du registre logique est testée. Les instructions qui apparaissent souvent comme dernière instruction d'une portion de code ont une instruction équivalente ayant l'effet supplémentaire de stopper l'exécution. Nous en verrons des exemples dans la section suivante (MK1STOP, MK2STOP, POSTSTOP, ...).

II.5-Exemples de fragments de code de la machine

Nous donnons ci-dessous le code créé par le compilateur METAL pour les fonctions de constructions d'arbres associées à certaines des règles qui ont été décrites tout au long de cet article. Notons que, sur chaque ligne de code, seule l'information qui précède le symbole «:» est utilisée par la machine, le reste de la ligne étant un désassemblage de ce code sous une forme plus lisible.

```
<COULEUR>          ::= %COULEUR ;
    couleur-atom(%COULEUR)

    34   1                 :     GPS          COULEUR
     4                     :     STOP

<FLIP>             ::= pivoter <FLIP> ;
    pivot(<FLIP>)

     7                     :     INI1
    37   12                :     MK1STOP      PIVOT
```

```
<FLIP>          ::= en <COULEUR> encadrer <FLIP> fincadre ;
    cadre(<COULEUR>,<FLIP>)

     8                      :    INI2
    38  19                  :    MK2STOP    CADRE

<LISTE_LIGNES>  ::= <LIGNE> ;
    lignes-list((<LIGNE>))

     7                      :    INI1
    40  28                  :    MKLSTOP    LIGNES

<LISTE_LIGNES>  ::= <LISTE_LIGNES> <LIGNE> ;
    lignes-post(<LISTE_LIGNES>,<LIGNE>)

     8                      :    INI2
    46   1                  :    POSTSTOP       1

<FLIP>          ::= horiz <LISTE_BANDES> fhoriz ;
    <LISTE_BANDES>

     4                      :    STOP

<FLIP>          ::= vertic <LISTE_BANDES> fvertic ;
    vertic-list(<LISTE_BANDES>)

     7                      :    INI1
    35  26                  :    RNLIST     VERTIC
     4                      :    STOP

<TEXTE_OU_CADRE>    ::= ecrire <FLIP> ;
    fonte(couleur,<FLIP>)

     7                      :    INI1
    12   1                  :    MKO        COULEUR
    38  17                  :    MK2STOP    FONTE

<FLIP>              ::= en <COULEUR> <TEXTE_OU_CADRE> ;
    case <TEXTE_OU_CADRE>
        when fonte(X,Y)
            => fonte(<COULEUR>,Y)
        when cadre(X,Y)
            => cadre(<COULEUR>,Y)
    end case

    30   2                  :    INI            2
    20   2                  :    PD             2
    28   1                  :    LOAD           1
```

```
    33   1                :    MATCH          1
    27   3                :    JPF            3
    29   2                :    ETIQ           *2*
    18   2                :    GIMETA     Y
    20   1                :    PD             1
    14  17                :    MK2        FONTE
    25   1                :    JP             1
    29   3                :    ETIQ           *3*
    33   2                :    MATCH          2
    27   5                :    JPF            1
    29   4                :    ETIQ           *4*
    18   2                :    GIMETA     Y
    20   1                :    PD             1
    14  19                :    MK2        CADRE
    25   1                :    JP             1
    29   1                :    ETIQ           *1*
     4                    :    STOP
```

```
<EN_COULEUR>::= en <COULEUR> ;
    fonte(<COULEUR>,null_tree)
```

```
    30   1                :    INI            1
    12   7                :    MKO        NULL_TREE
    20   1                :    PD             1
    38  17                :    MK2STOP    FONTE
```

```
<FLIP>       ::= <EN_COULEUR> ecrire <FLIP> ;
    let fonte(X,Y)=<EN_COULEUR> in
        fonte(X,<FLIP>)
```

```
    30   2                :    INI            2
    20   1                :    PD             1
    28   1                :    LOAD           1
    33   3                :    MATCH          3
    26   6                :    JPT            6
    32   1                :    CERR           1
    25   7                :    JP             7
    29   6                :    ETIQ           *6*
    20   2                :    PD             2
    18   1                :    GIMETA     X
    14  17                :    MK2        FONTE
    29   7                :    ETIQ           *7*
     4                    :    STOP
```

Remerciements

Les travaux effectués quotidiennement par G. Kahn et B. Lang pour maintenir et améliorer le système MENTOR, ainsi que leurs conseils et leurs idées, ont rendus

possible une implémentation propre du langage Métal. La mise au point de ce langage doit beaucoup au travail de E. Morcos qui, en développant MENTOR-Ada, a testé ce langage de façon approfondie. Ce papier a été photo-composé sur la VIP de l'I.N.R.I.A. en utilisant le système mis au point par G. Cousineau et G. Huet.

Appendice: le programme Flip.Métal

Nous présentons ci-dessous le programme Métal qui définit MENTOR-Flip. Ce programme est ordonné en chapitres et en sections («rules» et «abstract syntax»). Cette structure a comme unique objet de rendre le programme plus lisible et plus manipulable mais n'introduit aucun nouvel élément dans la logique du programme. Le nom des chapitres, par exemple, n'a d'autre importance que documentaire. De même, la répartition des règles dans les différentes sections «rules», ainsi que la répartition de la syntaxe abstraite dans les sections «abstract syntax» n'a aucune répercussion sur la logique du programme, c'est à dire sur le langage défini par ce programme. Par contre, l'en-tête du programme donne le nom qui sera le nom officiel du nouveau langage sous MENTOR.

Le dernier chapitre de ce programme définit les points d'entrée de l'analyseur qui existeront en MENTOR-Flip. Rappelons en effet que, en MENTOR, tout fragment de programme entré à la console est toujours analysé syntaxiquement et transformé en sa représentation arborescente. L'existence de plusieurs points d'entrée dans l'analyseur et le constructeur est donc indispensable pour une utilisation efficace et agréable du système. Les noms des points d'entrée, qui doivent être des noms de phylums déclarés dans une section «abstract syntax», sont donnés entre crochets en tête de la partie droite des productions. La première production, qui introduit le non-terminal <FLIP_AXIOME>, sert uniquement à pouvoir définir proprement ces points d'entrée.

Dans la section «abstract syntax» du chapitre «ATOMES» on peut voir, en partie droite des définitions d'opérateurs nullaires, des expressions du genre «implemented as IDENTIFIER». Ces expressions sont optionnelles et permettent à MENTOR d'optimiser l'implémentation des différents types d'atomes intervenant dans le langage.

Bien qu'aucun exemple ne le mette en évidence dans ce programme, il est possible d'introduire des commentaires dans un programme Métal: toute ligne dont le premier caractère non blanc est le caractère «*» est considérée par Métal comme une ligne de commentaires.

En regardant bien, vous trouverez dans le programme ci-dessous un phylum «COMMENT», et les opérateurs «comment» et «comment_s». Ils définissent les arbres qui seront construits pour les commentaires du langage objet, Flip. Dans la version actuellement en service du système, le traitement des commentaires est entièrement générique, c'est à dire que l'utilisateur n'a pas à s'en préoccuper dans son programme Métal. Il lui sera bien entendu possible de manipuler les commentaires de son langage avec les commandes standard de MENTOR prévues à cet effet. Nous prévoyons ultérieurement d'ajouter des instructions au langage Métal pour donner la possibilité de controler l'accrochage des commentaires dans les arbres.

Le caractère «#», qui apparaît dans la partie droite de la production qui définit le non-terminal <BANDE>, sert à forcer le caractère «:» car celui-ci est un caratère réservé du langage Métal.

```
definition of FLIP is
    rules
        <FLIP_AXIOME>    ::= <FLIP> ;
            <FLIP>
        <FLIP>           ::= <COULEUR> ;
            <COULEUR>
        <FLIP>           ::= <CHAINE> ;
            <CHAINE>
        <FLIP>           ::= <LISTE_LIGNES> ;
            <LISTE_LIGNES>
        <FLIP>           ::= en <COULEUR> ecrire <FLIP> ;
            fonte(<COULEUR>,<FLIP>)
        <FLIP>           ::= en <COULEUR> encadrer <FLIP> fincadre ;
            cadre(<COULEUR>,<FLIP>)
        <FLIP>           ::= pivoter <FLIP> ;
            pivot(<FLIP>)
        <FLIP>           ::= horiz <LISTE_BANDES> fhoriz ;
            <LISTE_BANDES>
        <FLIP>           ::= vertic <LISTE_BANDES> fvertic ;
            vertic-list(<LISTE_BANDES>)
        <FLIP>           ::= couvrir <LISTE_FLIP> fcouvrir ;
            <LISTE_FLIP>
    abstract syntax
        FLIP             ::=couleur chaine lignes fonte
                             cadre pivot horiz vertic flips;

        fonte            -> COULEUR FLIP;
        cadre            -> COULEUR FLIP;
        pivot            -> FLIP;
        horiz            -> BANDE + ... ;
        vertic           -> BANDE + ... ;

    chapter LISTES
        rules
            <LISTE_FLIP>     ::= <FLIP> ;
                flips-list((<FLIP>))
            <LISTE_FLIP>     ::= <LISTE_FLIP> avec <FLIP> ;
                flips-post(<LISTE_FLIP>,<FLIP>)
            <LISTE_BANDES>   ::= <BANDE> ;
                horiz-list((<BANDE>))
            <LISTE_BANDES>   ::= <LISTE_BANDES> <BANDE> ;
                horiz-post(<LISTE_BANDES>,<BANDE>)
            <BANDE>          ::= <NOMBRE> #: <FLIP> ;
                bande(<NOMBRE>,<FLIP>)
            <LISTE_LIGNES>   ::= <LIGNE> ;
                lignes-list((<LIGNE>))
            <LISTE_LIGNES>   ::= <LISTE_LIGNES> <LIGNE> ;
                lignes-post(<LISTE_LIGNES>,<LIGNE>)
        abstract syntax
            flips            -> FLIP + ... ;
            lignes           -> LIGNE + ... ;
            bande            -> NOMBRE FLIP;
            comment_s        -> COMMENT + ... ;
```

```
            BANDE               ::=bande;
            COMMENT             ::=comment;
    end chapter;

    chapter ATOMES
        rules
            <COULEUR>           ::= %COULEUR ;
                couleur-atom(%COULEUR)
            <CHAINE>            ::= %CHAINE ;
                chaine-atom(%CHAINE)
            <NOMBRE>            ::= %NOMBRE ;
                nombre-atom(%NOMBRE)
            <LIGNE>             ::= %LIGNE ;
                ligne-atom(%LIGNE)
        abstract syntax
            COULEUR             ::=couleur;
            CHAINE              ::=chaine;
            NOMBRE              ::=nombre;
            LIGNE               ::=ligne;

            couleur             -> implemented as IDENTIFIER;
            chaine              -> implemented as STRING;
            nombre              -> implemented as INTEGER;
            ligne               -> implemented as STRING;
            comment             -> implemented as STRING;
    end chapter;

    chapter POINTS_D_ENTREE
        rules
            <FLIP_AXIOME>       ::= [FLIP] <FLIP> ;
                <FLIP>
            <FLIP_AXIOME>       ::= [BANDE] <BANDE> ;
                <BANDE>
            <FLIP_AXIOME>       ::= [LIGNE] <LIGNE> ;
                <LIGNE>
            <FLIP_AXIOME>       ::= [COULEUR] <COULEUR> ;
                <COULEUR>
            <FLIP_AXIOME>       ::= [CHAINE] <CHAINE> ;
                <CHAINE>
            <FLIP_AXIOME>       ::= [NOMBRE] <NOMBRE> ;
                <NOMBRE>
    end chapter;
end definition
```

Références bibliographiques

BSTJ 78 UNIX Time-Sharing System, The Bell System Technical Journal, Vol 57, No 6, Part 2, July-August 1978

DON 75 V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang, J.J. Levy, «A structure oriented program editor: a first step toward computer assisted programming», International Computing Symposium, North Holland Publishing Co, (1975)

DON 79 V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang, «Introduction au système MENTOR et à ses applications», Journées francophones sur la certification du logiciel, Genève, Janvier 1979

DON 80 V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang, «Programming environments based on structured editors: The Mentor experience», I.N.R.I.A. Rapport de recherche No. 26, Juillet 1980

KAHN 81 G. Kahn, «FLIP, Manuel d'utilisation» I.N.R.I.A., Rapport technique No. 2, Juin 1981.

MEL 80 B. Mélèse, «Manipulation de programmes Pascal au niveau des concepts du langage», Thèse de 3ème cycle, Université Paris XI Orsay (1980)

MEL 81 B. Mélèse, «Mentor, L'environnement Pascal» I.N.R.I.A., Rapport technique no. 5, Octobre 1981.

MEL 82 B. Mélèse, «Métal, manuel d'utilisation» publication prochaine. Manuscript disponible à l'I.N.R.I.A

MOR 79 E. Morcos, «Etude des effets de bord des appels de procédures et de fonctions dans le langage Pascal», Thèse de 3ème cycle, Université Paris XI Orsay (1979).

MOS 78 P. Mosses, «SIS: A compiler generator system using denotational semantics» DAIMI, University of Aarhus, 1978

MUL 80 «The Mentor Program Manipulation System», Disponible dans la documentation du système MULTICS de l'I.N.R.I.A.

SCH 82 A. Schroeder, «Outils de mesures de programmes Pascal -Manuel d'utilisation-» I.N.R.I.A., rapport technique no. 10 Avril 1982

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique